Scientific Computing Accelerated on FPGA 2022

FPGA compute acceleration with Intel® oneAPI

Maurizio Paolini

Field Applications Engineer, Intel Corporation

July 2022

Copyright © 2022 Intel Corporation.

This document is intended for personal use only.

Unauthorized distribution, modification, public performance,

public display, or copying of this material via any medium is strictly prohibited

Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Agenda

FPGA Compute Acceleration with Intel® oneAPI

- Introduction to oneAPI
- The DPC++ programming language
- oneAPI Development Flow for FPGAs
- FPGA hardware for oneAPI
- Introduction to Intel® DevCloud

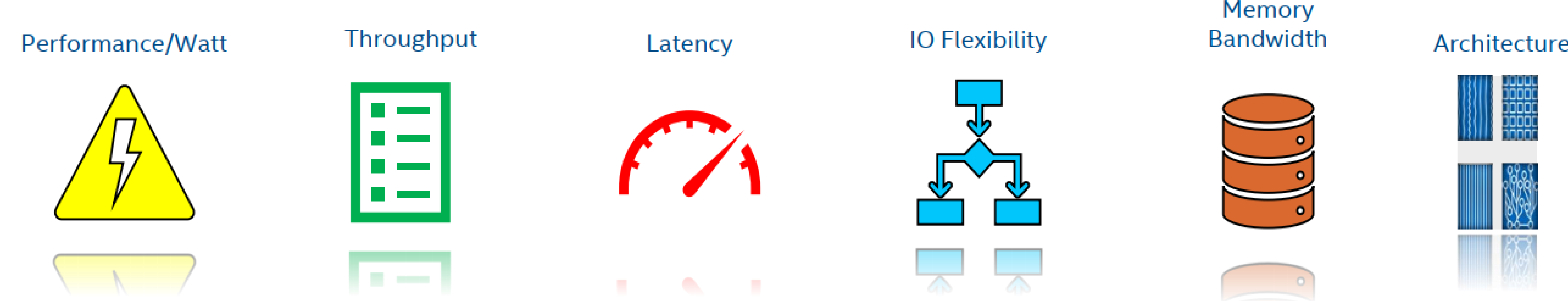
Copyright © 2022 Intel Corporation

Introduction to oneAPI

Advantages of Heterogeneous Computing

Multiple Architectures

- Developers can optimize specialized inline and offload workloads to meet business needs.
  - Strengths of individual xPUs (CPU, GPU, FPGAs, etc.) can be combined for the benefit of the overall system

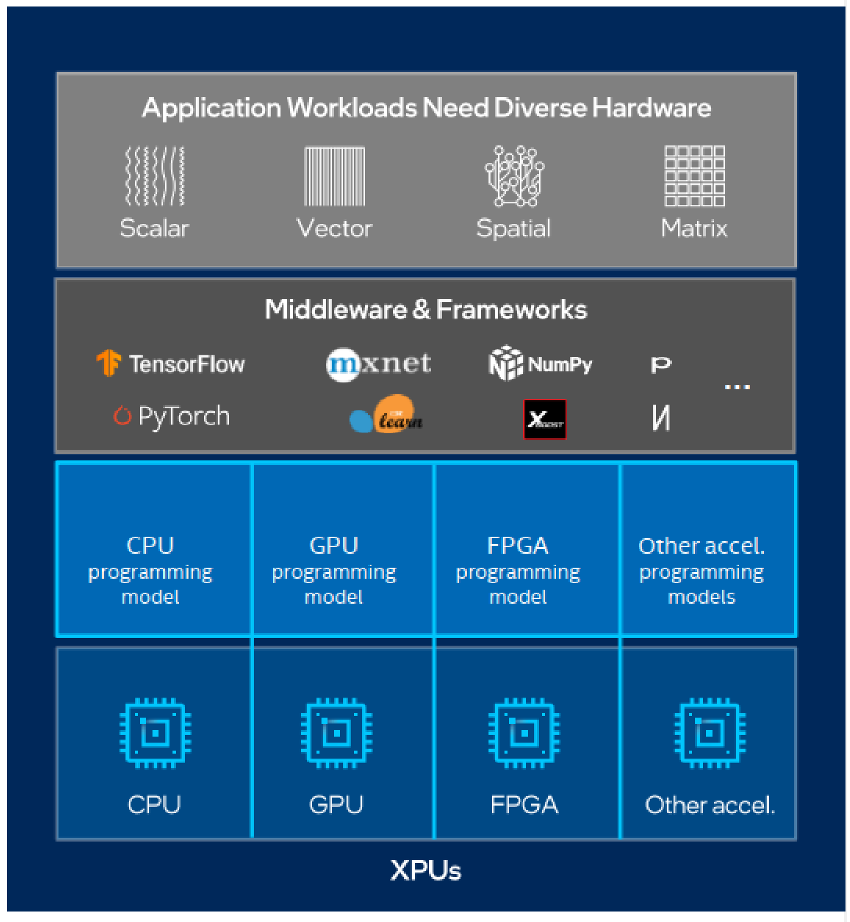


Copyright © 2022 Intel Corporation

Programming Challenges

Multiple Architectures

- **Separate** programming models and toolchains for each architecture.
  - Required **training** and **licensing** – compiler, IDE, debugger, analytics/monitoring tool, deployment tool, et al. – per architecture.
  - Challenging experience in **debug**, **monitoring**, and **maintenance** of a cross-architectural source code.
  - Difficult integration across proprietary IPs and architectures and no code re-use.
- Software development complexity **limits** freedom of architectural choice.
  - Isolated investments required for technical expertise to overcome the barrier-to-entry



Copyright © 2022 Intel Corporation

A Unified Programming Model

Multiple Architectures

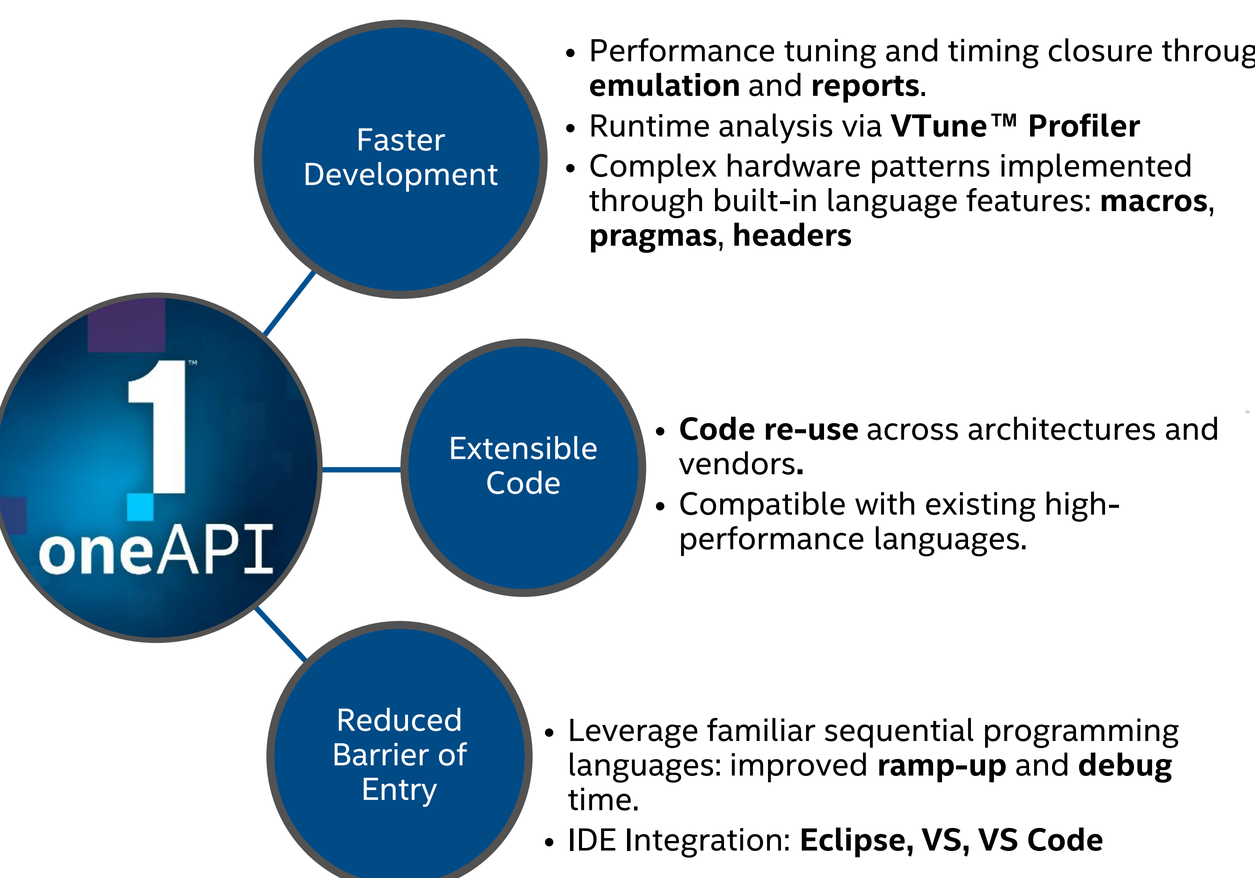
The **oneAPI** product delivers a unified programming model to simplify development across diverse architectures.

It guarantees:

- **Common developer experience** across Scalar, Vector, Matrix and Spatial architectures (CPU, GPU, AI and FPGA)
- Uncompromised native high-level language **performance**
- Industry standardization and open specifications

Copyright © 2022 Intel Corporation

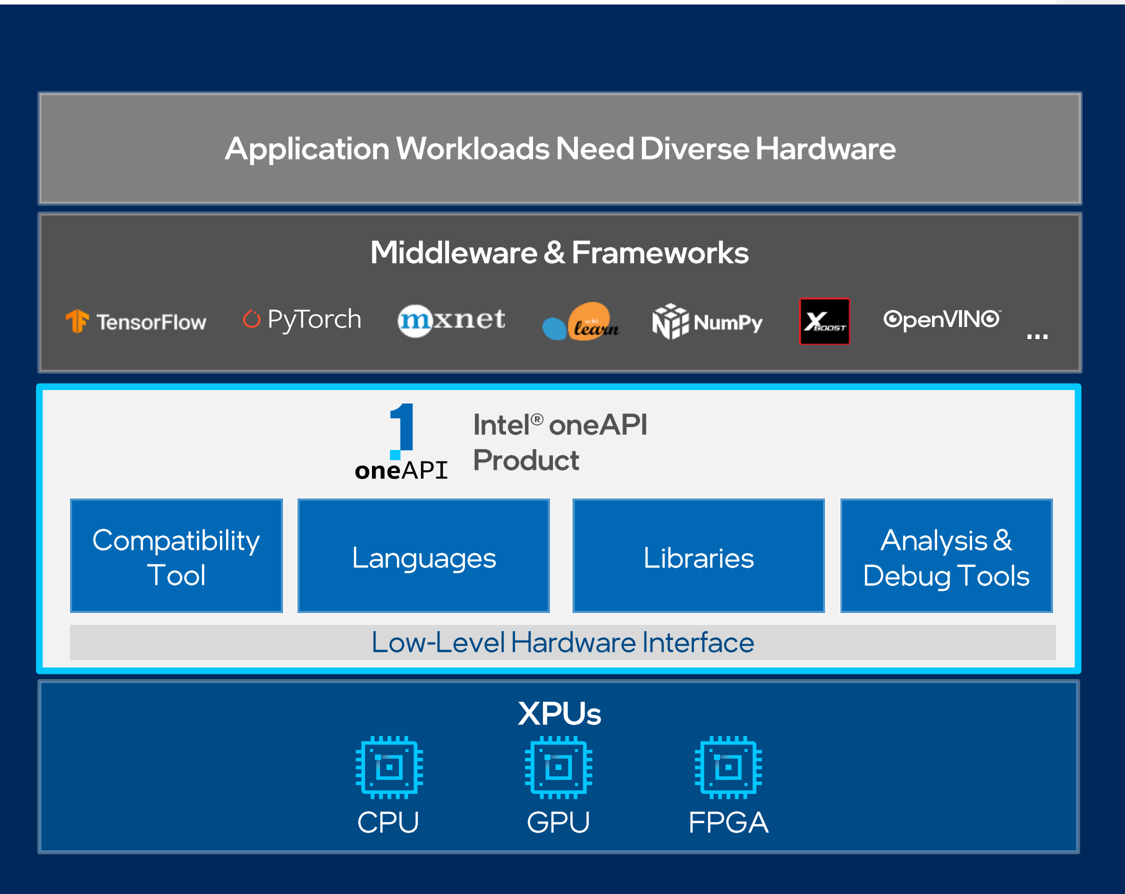
Intel® oneAPI Product



- Performance tuning and timing closure through **emulation** and **reports**.
- Runtime analysis via **VTune™ Profiler**
- Complex hardware patterns implemented through built-in language features: **macros**, **pragmas**, **headers**

- **Code re-use** across architectures and vendors**.**
- Compatible with existing high-performance languages.

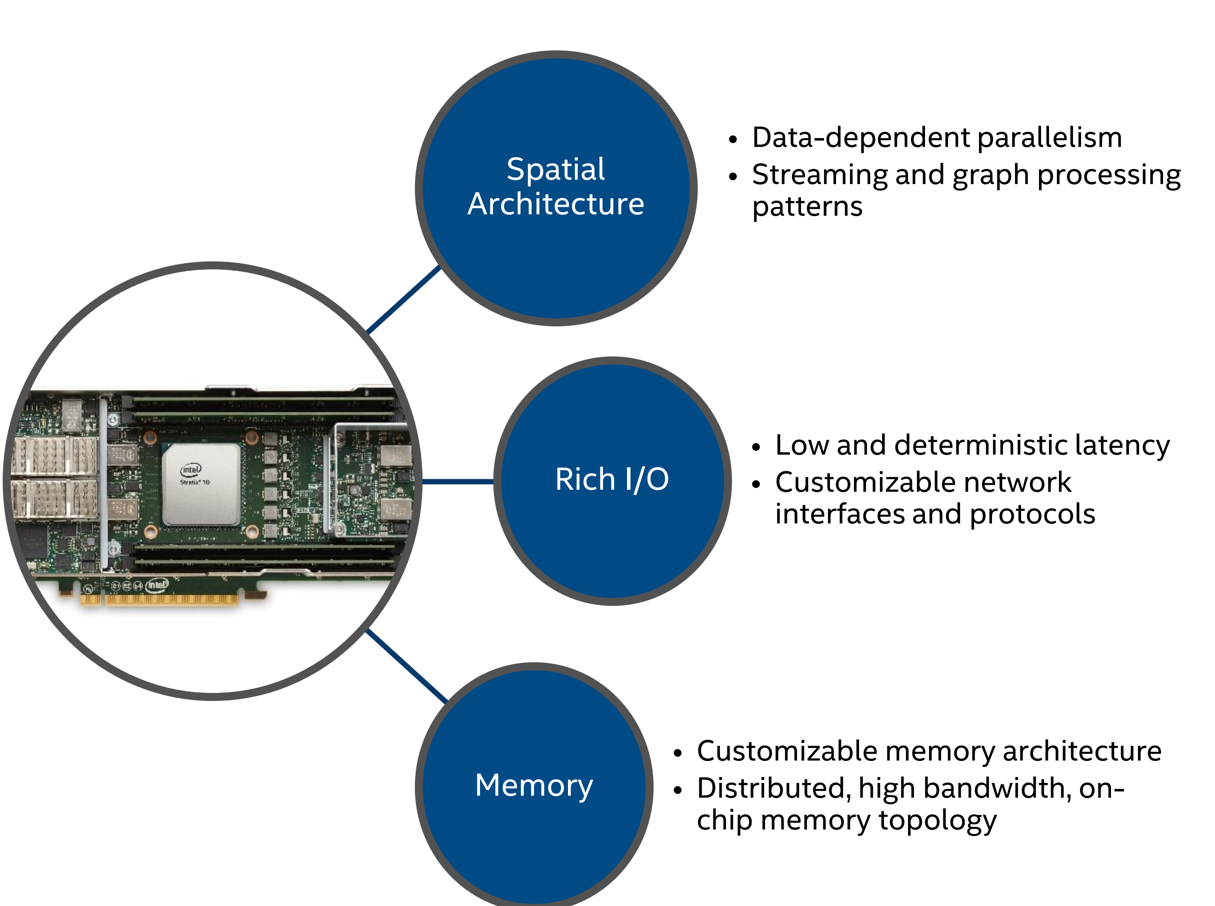
- Leverage familiar sequential programming languages: improved **ramp-up** and **debug** time.
- IDE Integration: **Eclipse, VS, VS Code**



Available Now

Copyright © 2022 Intel Corporation

Intel® FPGAs + Intel® oneAPI Toolkits

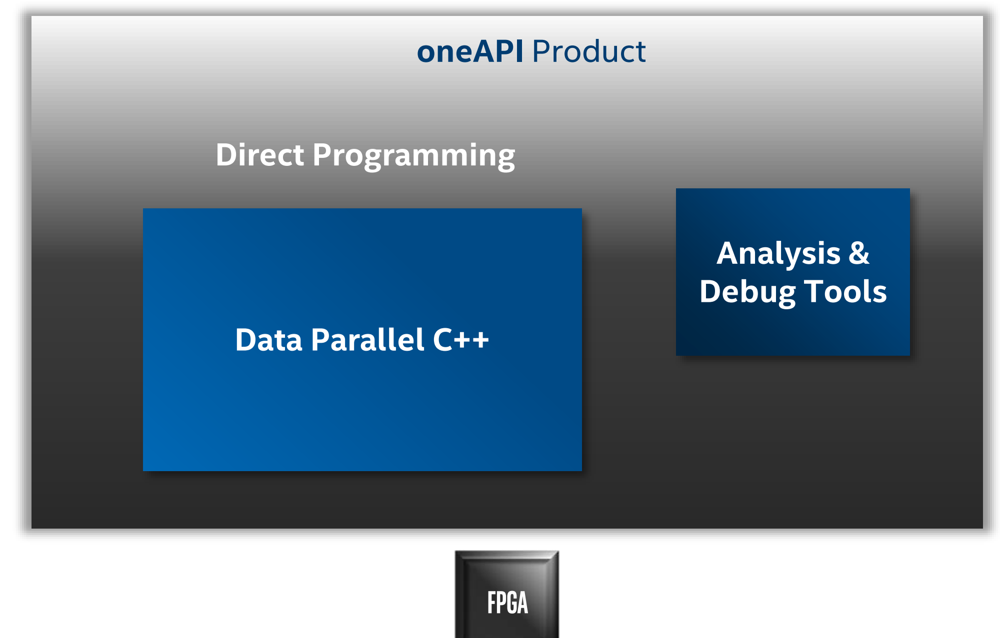


- Data-dependent parallelism
- Streaming and graph processing patterns

- Low and deterministic latency
- Customizable network interfaces and protocols

- Customizable memory architecture
- Distributed, high bandwidth, on-chip memory topology

+



Copyright © 2022 Intel Corporation

Large number of use cases

Examples

- Data compression
- Image compression
- File parsing
- Data Base acceleration
- Genomics
- Financial
- …

Accelerating Re-Pair Compression using FPGAs

FPGA-Accelerated Compression of Integer Vectors

PipeJSON: Parsing JSON at Line Speed on FPGAs

Resource-Efficient Database Query Processing on FPGAs

The DPC++ Programming Language

Data Parallel C++ (DPC++)

- Common language designed to target any XPU
  - Tuning still needed for each architecture
- Goal: to incorporate everything needed to get the best performance out of every architecture

Based on C++ and SYCL

- SYCL is based on OpenCL
- Think of it as SYCL + extensions

Allows for single-source targeting of accelerators

- Doesn't require multiple files

Open specification

Copyright © 2022 Intel Corporation

DPC++: Three Scopes

- DPC++ programs consist of 3 scopes:
  - Application scope - Code executed on the host
  - Command group scope - Code for submitting data and commands to the accelerator
  - Kernel scope – Code executed on the accelerator
- The full capabilities of C++ are available at application and command group scope
- At kernel scope there are limitations in accepted C++
  - Most important is no recursive code
  - See SYCL specification for complete list

```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Application Scope

Command Group Scope

Kernel Scope

Copyright © 2022 Intel Corporation

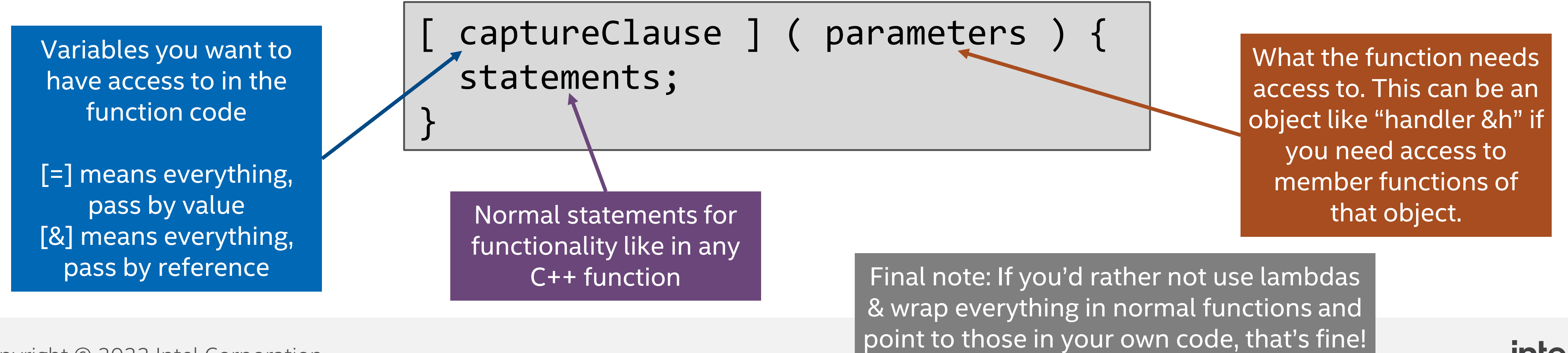
The “Runtime”

- The DPC++/SYCL runtime is the program running in the background on the host controlling the execution and data passing needs of the heterogeneous compute execution
- It handles:
  - Kernel and host execution in an order imposed by data dependency needs (discussed later)
  - Passing data back and forth between the host and device
  - Querying the device
  - Etc.

Copyright © 2022 Intel Corporation

A Note About Lambda Functions

- Two common constructs in DPC++ - queue submissions and kernel dispatch functions - take function pointers as arguments
- This doesn't lend itself to simple, in-line code
- To write simpler and neat code, lambda functions are used
- Lambda functions are un-named functions used in-line with other code
- If you are not familiar with them, here is a simple guide



Copyright © 2022 Intel Corporation

DPC++ Class: device

- The device class represents the accelerators in a oneAPI system
- The device class contains member functions for querying information about the available devices
- The function get_info gives information about a device:
  - Name, vendor, and version of the device
  - Width for built in types, clock frequency, cache width and sizes, online or offline

```
// Get all of the devices a system is capable of operating
std::vector<device> my_devices = device::get_devices();
// Grab the first device to print info out for
device my_device = my_devices[0];
// Print the name of the first device
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```

Copyright © 2022 Intel Corporation

DPC++ Class: device_selector

- The device_selector enables the selection of a device to execute kernels on
- Use the selector when you create a queue (covered next)
- The code sample shows use of several example device selectors, including an FPGA
- A custom device selector can be defined and used for targeting specific devices

```
// Other example selectors that are not FPGAs
// default_selector selector;
// host_selector selector;
// cpu_selector selector;
// gpu_selector selector;

// Create the selector as an fpga_selector type
INTEL::fpga_selector selector;

// Use the selector when you create a queue
queue q(selector);
```

Copyright © 2022 Intel Corporation

DPC++ Class: queue

- A queue is a mechanism where work is submitted to a device
- A queue submits command groups to be executed by the SYCL runtime
- A queue.submit() is the beginning of the command group scope
  - Groups of work to be executed by the SYCL runtime on an accelerator
- A queue maps to a single device

```
// Declare a queue to a device
queue q(selector);

// Submit things to the queue
q.submit([&](handler& h) {
    // COMMAND GROUP CODE
});
```

The handler is a class that contains all of the command group functions of SYCL

You can think of it as an abstraction of the runtime

This keeps us from having to type handler:: again and again in the command group scope

Copyright © 2022 Intel Corporation

DPC++ Class: kernel

- The kernel encapsulates code that will be run on the accelerator
- A kernel object is not explicitly constructed by the user
- It is constructed when a kernel dispatch function, such as parallel_for() or single_task() is called

```
q.submit([&](handler& h) {

  // The “kernel” is everything after the kernel dispatch function
  h.single_task<VectorAdd>([=]() {

    // Everything inside here is “KERNEL SCOPE”
    for (int i = 0; i < kSize; ++i) {
            c[i] = a[i] + b[i];
    }
});
```

Copyright © 2022 Intel Corporation

Single Task Kernels

- single_task() kernels allow complex or lengthy datapaths to be built from custom hardware in FPGAs
- Useful to offload code with dependencies that are difficult to execute in a data parallel fashion
- Look like CPU code
  - Contain an outer loop to process all data
- Ideal for & recommended for FPGAs

CPU Implementation

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

FPGA Kernel Implementation

```
h.single_task([=](){
    for (int i=0; i < 1024; i++) {
       A[i] =  B[i] + C[i];
    }
});
```

DPC++ Class: buffer and accessor

- buffer
  - Encapsulates data in a SYCL application
  - Across both devices and host!
- accessor
  - Mechanism to access buffer data
  - Determines data dependencies in that order kernel executions (covered later)

```
int main() {
  … // Code to set up standard C++ vectors

  buffer buf_a(vector_a);
  buffer buf_b(vector_b);
  buffer buf_c(vector_c);

  queue q(selector);

  q.submit([&](handler& h) {
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);

    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
            c[i] = a[i] + b[i];
      }
    });
  });
```

Copyright © 2022 Intel Corporation

#Include Files

- oneAPI programs require the include of cl/sycl.hpp
- Programs targeting FPGAs require the include of cl/sycl/INTEL/fpga_extensions.hpp

```
// Always include these at the top of your program

#include <CL/sycl.hpp>
#include <CL/sycl/INTEL/fpga_extensions.hpp>
```

Copyright © 2022 Intel Corporation

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Step 1: Create a device selector targeting the FPGA

Copyright © 2022 Intel Corporation

```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

DPC++ Simple Program Walk-Through

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Copyright © 2022 Intel Corporation

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c, int N) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Copyright © 2022 Intel Corporation

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command group for execution

Copyright © 2022 Intel Corporation

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

Copyright © 2022 Intel Corporation

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

Step 6: Send a kernel for execution

Copyright © 2022 Intel Corporation

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  INTEL::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

Step 6: Send a kernel for execution

Done!

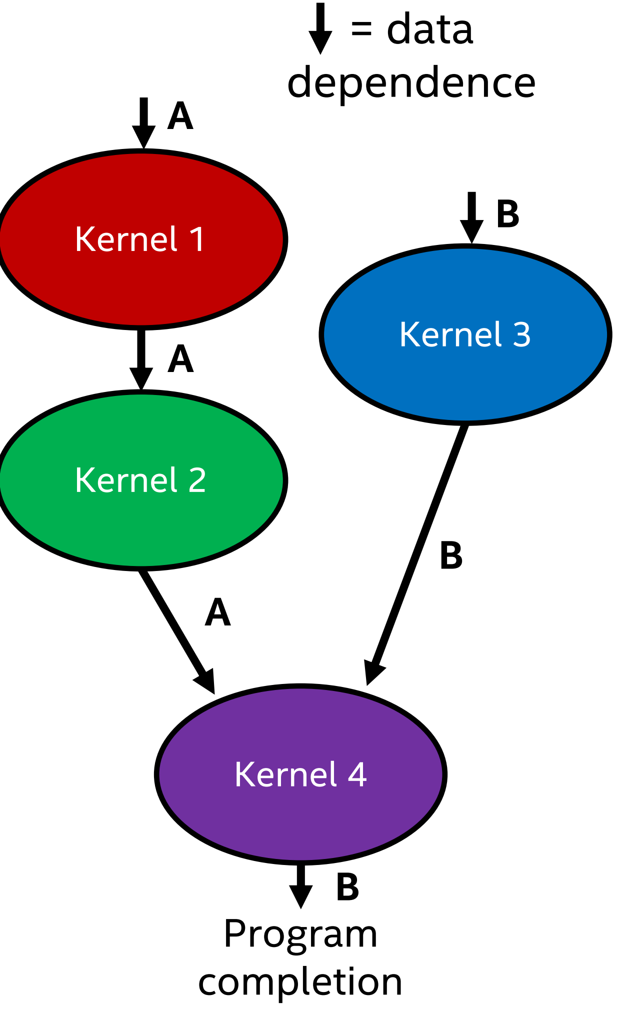
The contents of buf_c are copied to *c when the function finishes

(because of the buffer destruction of buf_c)

Copyright © 2022 Intel Corporation

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                    } Kernel 1

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                    } Kernel 2

  Q.submit([&](handler& h) {
    auto out = B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                    } Kernel 3

  Q.submit([&](handler& h) {
  auto in = A.get_access<access::mode::read>(h);
  auto inout =
    B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      inout[idx] *= in[idx]; }); });                } Kernel 4
}
```

Kernel Execution Order

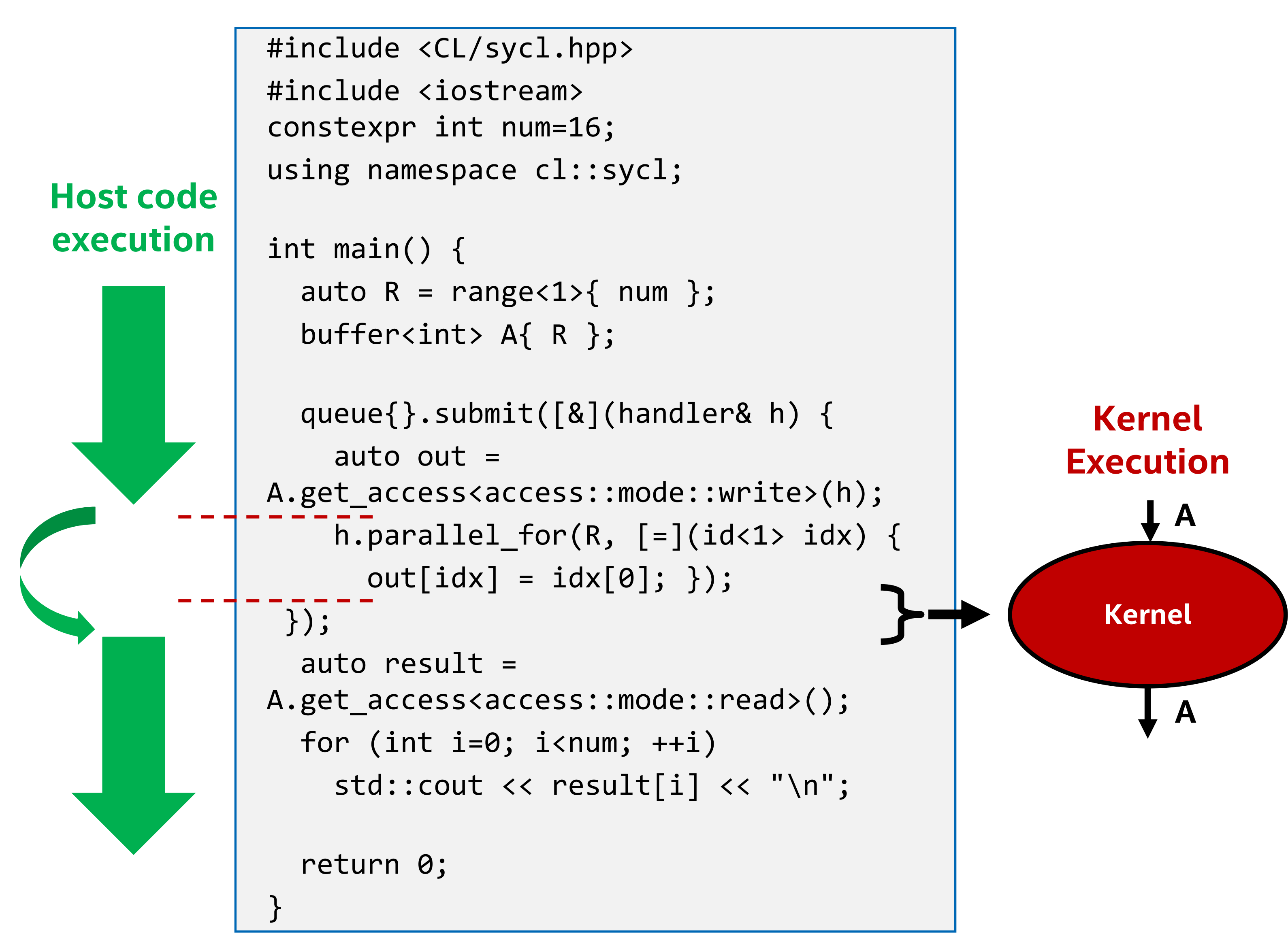


- Kernels can execute at the same time
  - If no data dependences
- Accessors are used to determine dependences
- Execution ordering is automatically determined

Copyright © 2022 Intel Corporation

Asynchronous Host/Kernel Execution

- The execution of the host code is asynchronous to what is being executed on the accelerator
- If you need synchronization, you must impose that yourself




```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R };

  queue{}.submit([&](handler& h) {
    auto out =
A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; });
 });
  auto result =
A.get_access<access::mode::read>();
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "\n";

  return 0;
}
```

Copyright © 2022 Intel Corporation

Synchronization Method 1: Host Accessor

- In the command scope, accessors are created for the accelerator
- In the application scope, accessors are created for the host
- A host accessor creates a dependency node in the execution graph
  - Execution at the host is blocked until the data is ready

```
int main() {
  constexpr int N = 100;
  auto R = range<1>(N);
  std::vector<double> v(N, 10);
  queue q;

  buffer<double, 1> buf(v.data(), R);
  q.submit([&](handler& h) {
  auto a = buf.get_access<access::mode::read_write>(h);
  h.parallel_for(R, [=](id<1> i) {
    a[i] -= 2;
    });
  });

  auto b = buf.get_access<access::mode::read>();
  for (int i = 0; i < N; i++)
    std::cout << v[i] << "\n";
  return 0;
}
```

Command Scope

Application Scope

Copyright © 2022 Intel Corporation

Synchronization Method 2: Buffer Destruction

- Buffer creation happens within a separate function scope
- When execution advances beyond this function scope, buffer destructor is invoked
- Relinquishes ownership of data and copies back the data to the host memory
- Scope can also be created with simple use of { }

```
#include <CL/sycl.hpp>
constexpr int N=100;
using namespace cl::sycl;

void dpcpp_code(std::vector<double> &v, queue &q){
    auto R = range<1>(N);
    buffer<double, 1> buf(v.data(), R);
    q.submit([&](handler& h) {
    auto a = buf.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> i) {
        a[i] -= 2;
        });
    });
}

int main() {
    std::vector<double> v(N, 10);
    queue q;
    dpcpp_code(v,q);
    for (int i = 0; i < N; i++)
        std::cout << v[i] << "\n";
    return 0;
}
```

Copyright © 2022 Intel Corporation

Unified Shared Memory

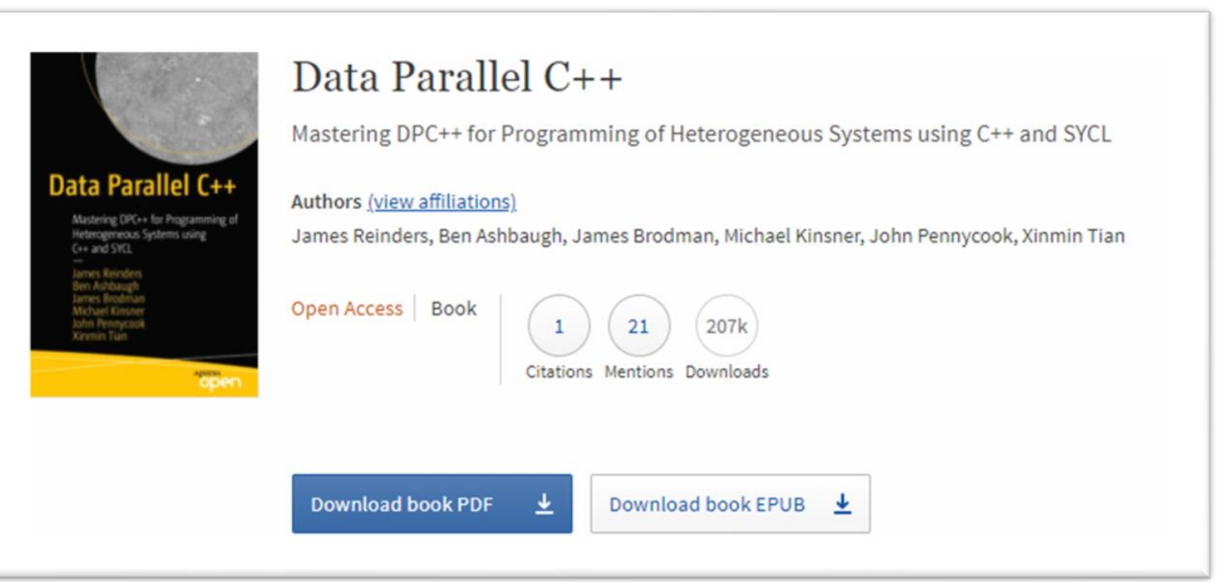
- USM => unified virtual address space
  - Any pointer value returned by a USM allocation routine is valid on the device
  - Three different types of allocations are defined
    - Device - data on device, must be explicitly transferred, not accessible from host
    - Host - data on host, accessible from device
    - Shared - data on host and/or on device, runtime managed data movement
  - Host allocation ideal for streaming applications at interface speed

```
562  q.submit([&](handler& h) {
563    h.single_task<paralleladdKernel>([=]() [[intel::kernel_args_restrict]] {
564
565      host_ptr<int> in(in_host);
566      host_ptr<int> out(out_host);
567
568      int i_cnt = 0;
569
570          // Init regs to zero
571          [[intel::fpga_register]] std::array<int, 16> raw_data;
572          #pragma unroll
573          for (int i = 0; i < 16; i++) { raw_data[i] = 0; }
574
575          [[intel::fpga_register]] std::array<int, 8> add_data;
576          #pragma unroll
577          for (int i = 0; i < 8; i++) { add_data[i] = 0; }
578
579
580
581        // Run over all CLs
582        while(i_cnt < iterations) {
583
584          // Load complete CL in one clock cycle, (same for PCIe and DDR4)
585          #pragma unroll
586          for (uint idx = 0; idx < 16; idx++) {
587            {
588              raw_data[idx] = in[idx + i_cnt*16];
589            }
590          }
591
592          add_data[0] = raw_data[0] + raw_data[8];
593          add_data[1] = raw_data[1] + raw_data[9];
594          add_data[2] = raw_data[2] + raw_data[10];
595          add_data[3] = raw_data[3] + raw_data[11];
596          add_data[4] = raw_data[4] + raw_data[12];
597          add_data[5] = raw_data[5] + raw_data[13];
598          add_data[6] = raw_data[6] + raw_data[14];
599          add_data[7] = raw_data[7] + raw_data[15];
600
601          // Write results back with half CL, as we can write and read
602          // a CL per clock cycle this will create no bottleneck
603          #pragma unroll
604          for (uint idx = 0; idx < 8; idx++) {
605            {
606              out[idx + i_cnt*8] = add_data[idx];
607            }
608          }
609
610
611          i_cnt++;
612
613        }
614
615    });
616  }).wait();
```

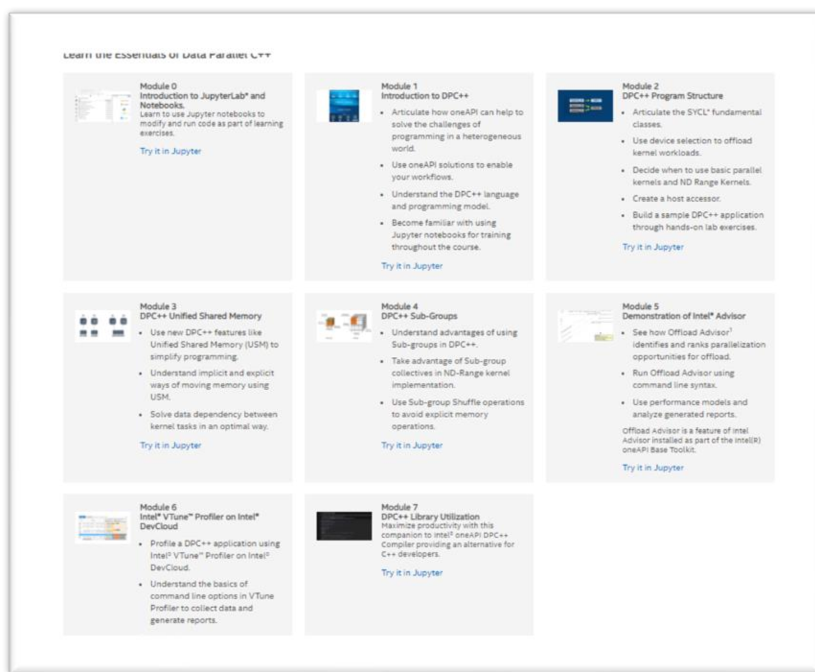
Copyright © 2022 Intel Corporation

Learn More About DPC++

- Download DPC++ book for free
  - https://link.springer.com/book/10.1007%2F978-1-4842-5574-2



- DPC++ Training Modules
  - https://devcloud.intel.com/oneapi/get_started/baseTrainingModules/



OneAPI Development Flow for FPGAs

Getting Started with oneAPI on an FPGA

Intel® oneAPI Base Toolkit

Intel® FPGA Add-on for oneAPI Base Toolkit

Board Support Package (BSP)

Note: Developers using custom platforms should download the Intel® FPGA Add-on for Intel® Custom Platforms with the respective Intel® Quartus® version and obtain a BSP from their 3rd part platform vendor.

Copyright © 2022 Intel Corporation

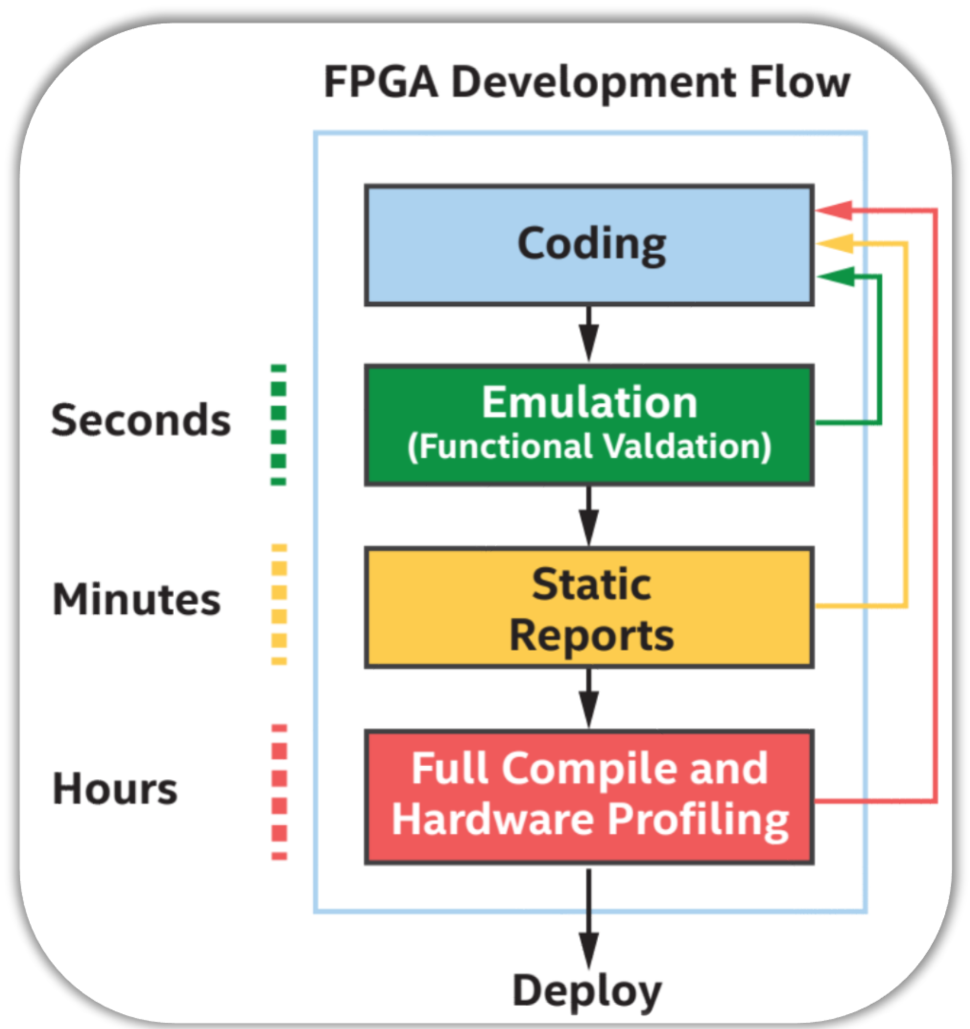
Installing oneAPI

- Get started by visiting the Intel® Software Developer Zone landing page for the Intel® oneAPI Toolkits
  - https://software.intel.com/en-us/oneapi
- Get the Intel® oneAPI Base Toolkit for Linux*
  - Supports compiles for emulation and the optimization report
- Install the Intel® FPGA Add-on for oneAPI Base Toolkit
  - Needed for compiles to FPGA hardware
  - Contains Intel® Quartus® Prime software “under the hood,” be sure to comply to required versions of operating system

Copyright © 2022 Intel Corporation

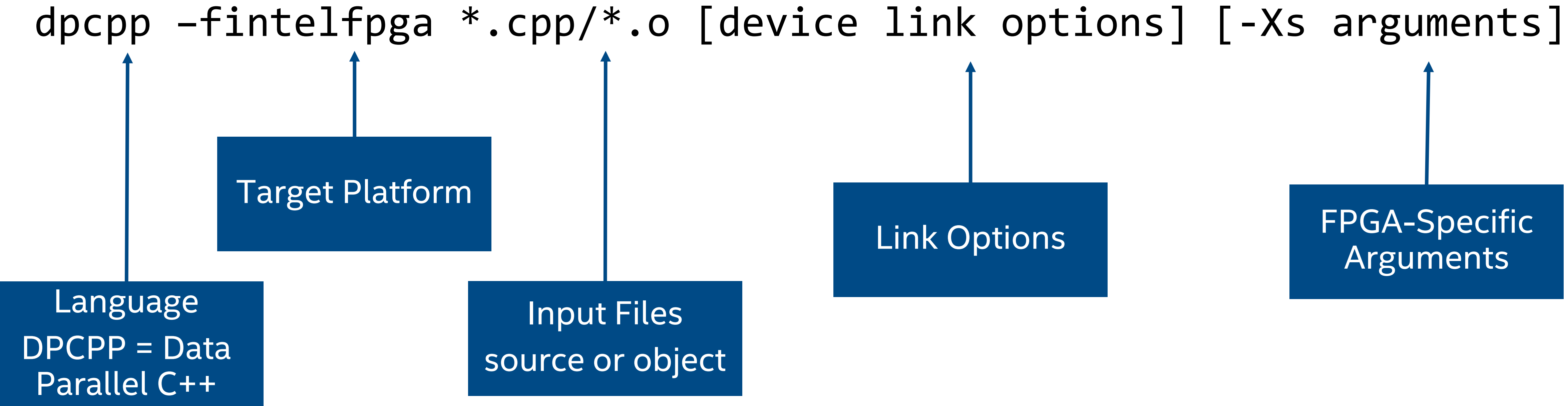
FPGA Development Flow for oneAPI Projects

- FPGA Emulator target (Emulation)
  - Compiles in seconds
  - Runs completely on the host
- Optimization report generation
  - Compiles in seconds to minutes
  - Identify bottlenecks
- FPGA bitstream compilation
  - Compiles in hours
  - Enable profiler to get runtime analysis



Copyright © 2022 Intel Corporation

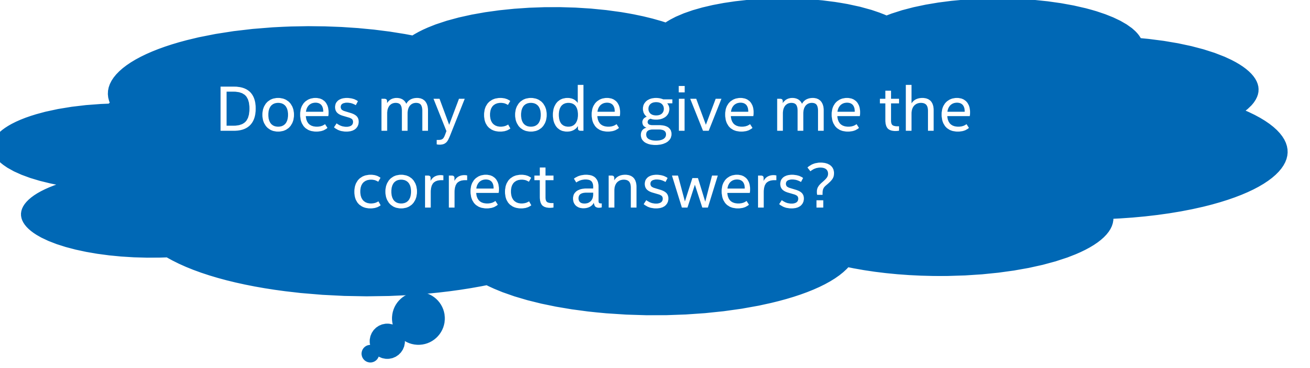
Anatomy of a dpcpp Command Targeting FPGAs



Copyright © 2022 Intel Corporation

Emulation

Seconds of Compilation



Quickly generate code that runs on the x86 host to emulate the FPGA

Developers can:

- Verify functionality of design through CPU compile and emulation.
- Identify quickly syntax and pointer implementation errors for iterative design/algorithm development.
- Enable deep, system-wide debug with Intel® Distribution for GDB.
- Functional debug of SYCL code with FPGA extensions.

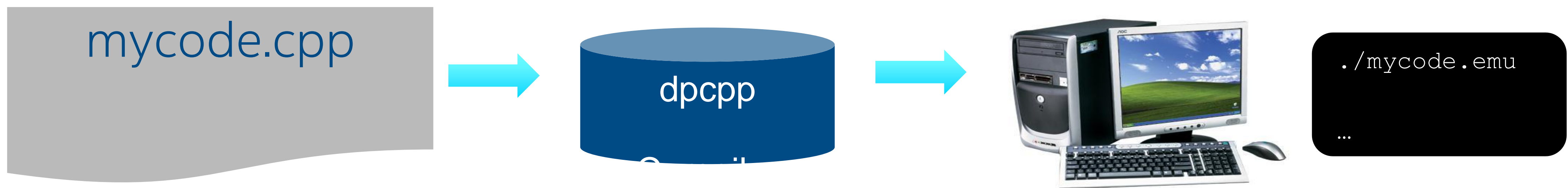
Copyright © 2022 Intel Corporation

Emulation Command

```
#ifdef FPGA_EMULATOR
  intel::fpga_emulator_selector device_selector;
#else
  intel::fpga_selector device_selector;
#endif
```

Include this construct in your code

```
dpcpp -fintelfpga <source_file>.cpp -DFPGA_EMULATOR
```



Copyright © 2022 Intel Corporation

Report Generation

Minutes of Compilation

Quickly generate a report to guide optimization efforts

Developers can:

- Identify any memory, performance, data-flow bottlenecks in their design.
- Receive suggestions for optimization techniques to resolve said bottlenecks.
- Get area and timing estimates of their designs for the desired FPGA.

Copyright © 2022 Intel Corporation

intel.

Command to Produce an Optimization Report

Two Step Method:

```
dpcpp -fintelfpga <source_file>.cpp -c -o <file_name>.o
dpcpp -fintelfpga <file_name>.o -fsycl-link -Xshardware
```

One Step Method:

```
dpcpp -fintelfpga <source_file>.cpp -fsycl-link -Xshardware
```

The default value for –fsycl-link is -fsycl-link=early which produces an early image object file and report

- A report showing optimization, area, and architectural information will be produced in <file_name>.prj/reports/

Copyright © 2022 Intel Corporation

Bitstream Compilation

Runs Intel Quartus Prime Software "under the hood" (no licensing required)

Developers can:

- Compile FPGA bitstream for their design and run it on an FPGA.
- Attain automated timing closure.
- Obtain in-hardware verification.
- Take advantage of Intel® VTune™ Profiler for real-time analysis of design.

Copyright © 2022 Intel Corporation

Compile to FPGA Executable with Profiler

Two Step Method:

```
dpcpp -fintelfpga <source_file>.cpp -c -o <file_name>.o
dpcpp -fintelfpga <file_name>.o –Xshardware -Xsprofile
```

One Step Method:

```
dpcpp -fintelfpga <source_file>.cpp –Xshardware -Xsprofile
```

The profiler will be instrumented within the image and you will be able to run the executable to return information to import into Intel® Vtune Amplifier.

To compile to FPGA executable without profiler, leave off –Xsprofile.

Copyright © 2022 Intel Corporation

Compiling FPGA Device Separately and Linking

- In the default case, the DPC++ Compiler handles generating the host executable, device image, and final executable
- It is sometimes desirable to compile the host and device separately so changes in the host code do not trigger a long compile

has_kernel.cpp

host_only.cpp

Partition code

This is the long compile

Then run this command to compile the FPGA image:

```
dpcpp -fintelfpga has_kernel.cpp -fsycl-link=image -o has_kernel.a -Xshardware
```

This command to produce an object file out of the host only code:

```
dpcpp -fintelfpga host_only.cpp -c -o host_only.o
```

This command to put the object files together into an executable:

```
dpcpp -fintelfpga has_kernel.a host_only.o -o executable.fpga
```

Copyright © 2022 Intel Corporation

Porting OpenCL code to oneAPI

- Same programming flow => easy porting
- Migration guidelines available at https://www.intel.com/content/www/us/en/develop/documentation/migrate-opencl-fpga-designs-to-dpcpp/top.html

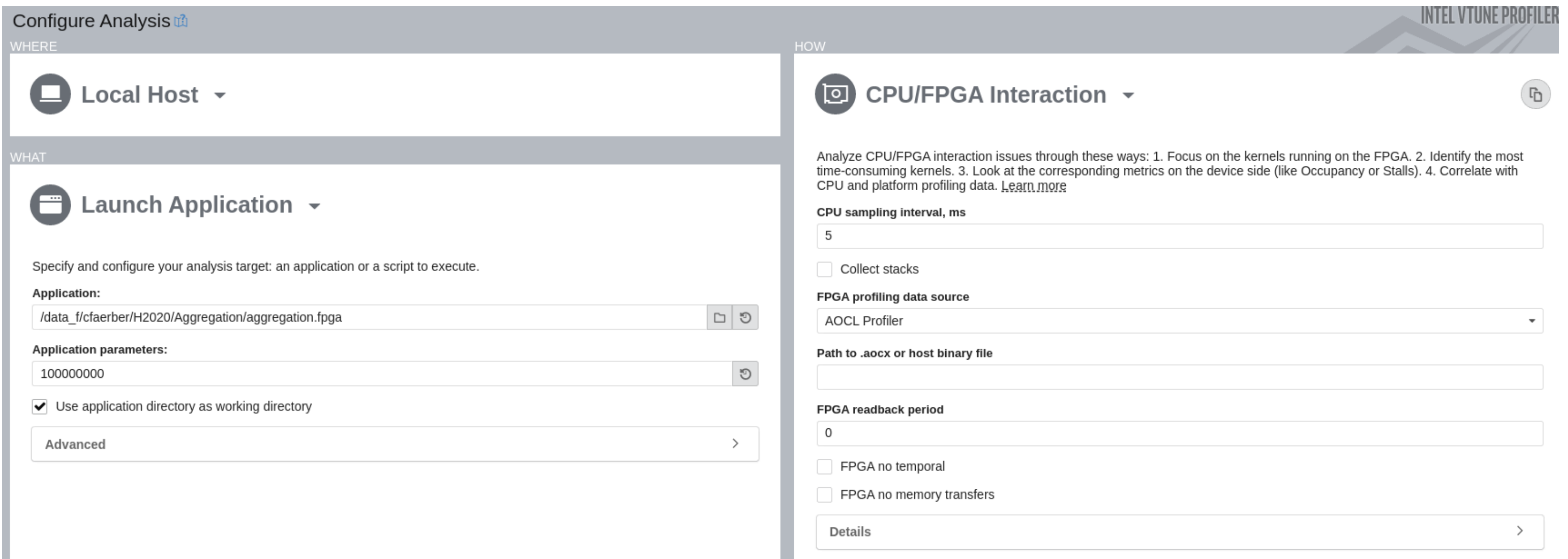
OpenCL	oneAPI
Kernel uses C99	Kernel uses DPC++
Autorun kernel	No autorun kernel but easy workaround available
Buffer read/write access is from the host point of view	Buffer access is from the device point of view
Buffers with clEnqueueWriteBuffer or USM buffers	Buffers and accessors controlled by SYCL runtime or USM pointers
Programming file AOCX separated from host executable	Single executable combining FPGA programming file and host executable – fat binary
VTune profiling only from host system	VTune profiling from host and kernel system

Copyright © 2022 Intel Corporation

Intel® VTune™

Start

- Add –Xsprofile to the set of flags in Makefile of hardware compile
- Start VTune with root rights and configure analysis CPU/FPGA

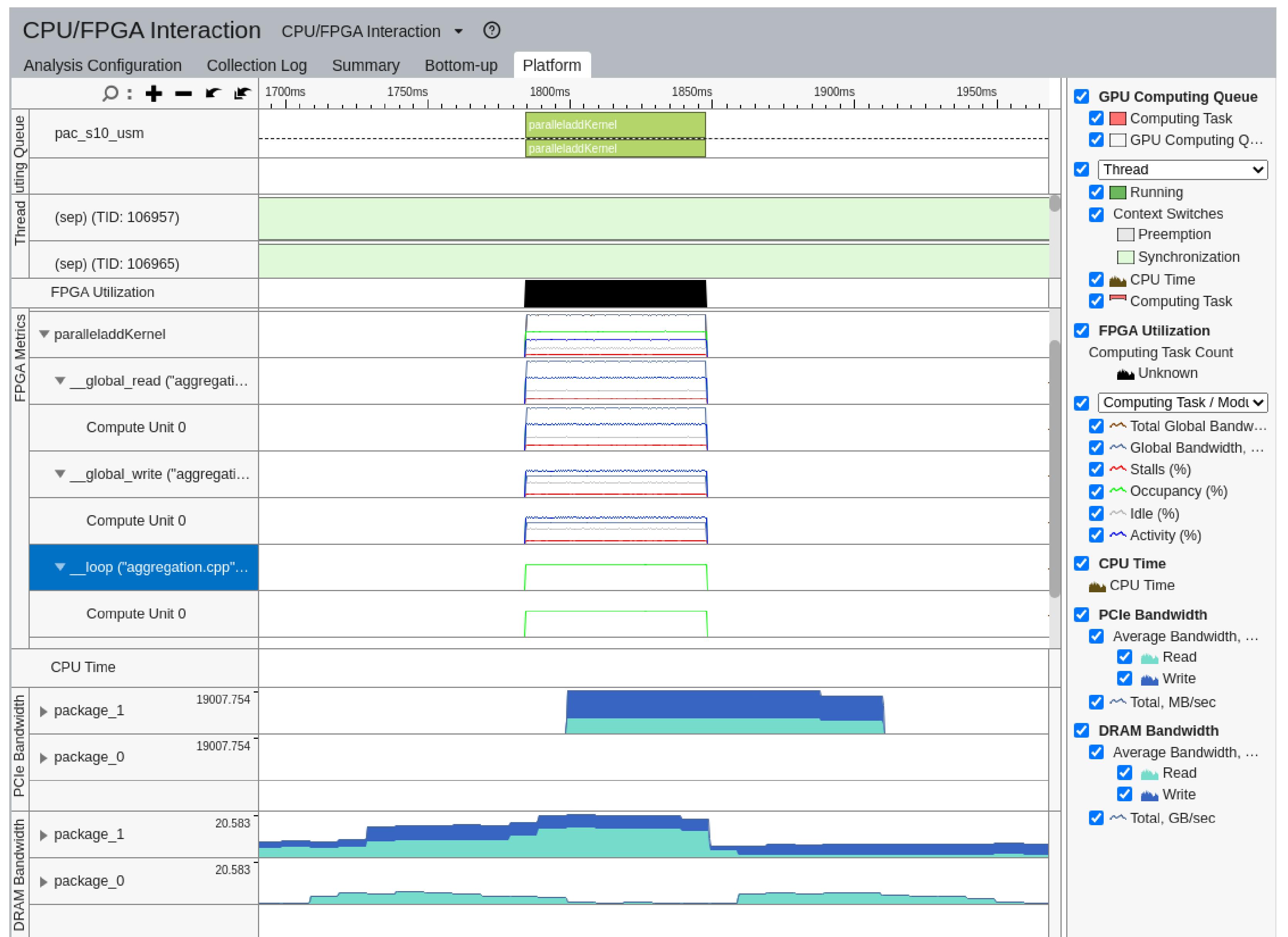


Copyright © 2022 Intel Corporation

intel.

Intel® VTune™

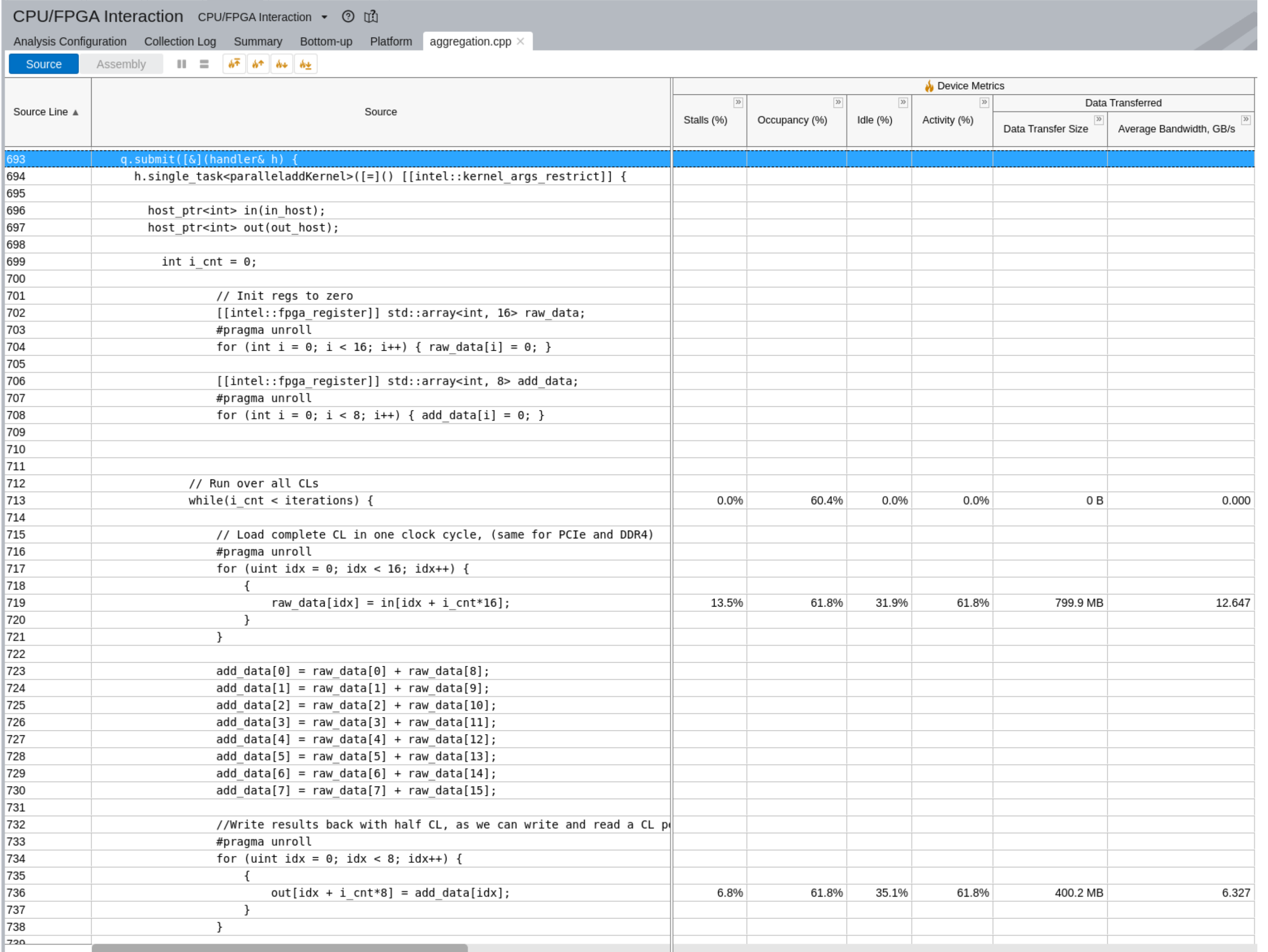
- Kernel schedule in respect to host
- FPGA utilization
- PCIe BW
- DRAM BW



Copyright © 2022 Intel Corporation

Intel® VTune™

- Performance counters inside kernel
- Bottleneck inside kernel visible
- Utilization of resources and throughput of each loop in kernel



Copyright © 2022 Intel Corporation

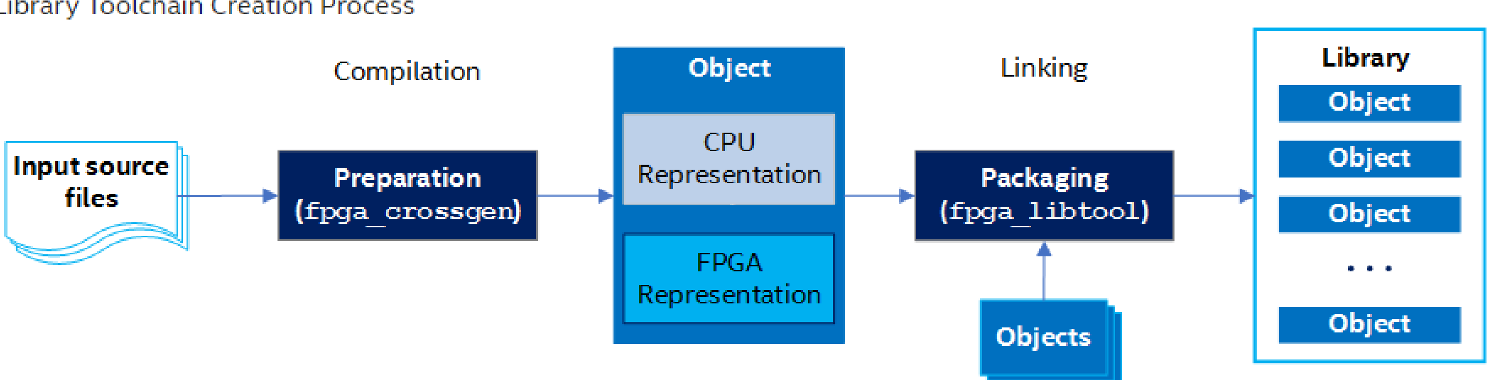
intel.

Use of RTL Libraries for FPGA in oneAPI

- Create a static library file using RTL
  - fpga_crossgen: rtl -> object
  - fpga_libtool: objects -> library

  Files needed:
    - RTL wrapper
    - XML description
    - Emulation model file (SYCL-based)

  RTL design constraints:
    - RTL module must have a clock port, a resetn port, and Avalon® streaming interface input and output ports
    - A single pair of ready and valid logic must control all the inputs
    - Declare the RTL module as stall-free if possible



- Include library file to use the functions inside your SYCL* kernels.

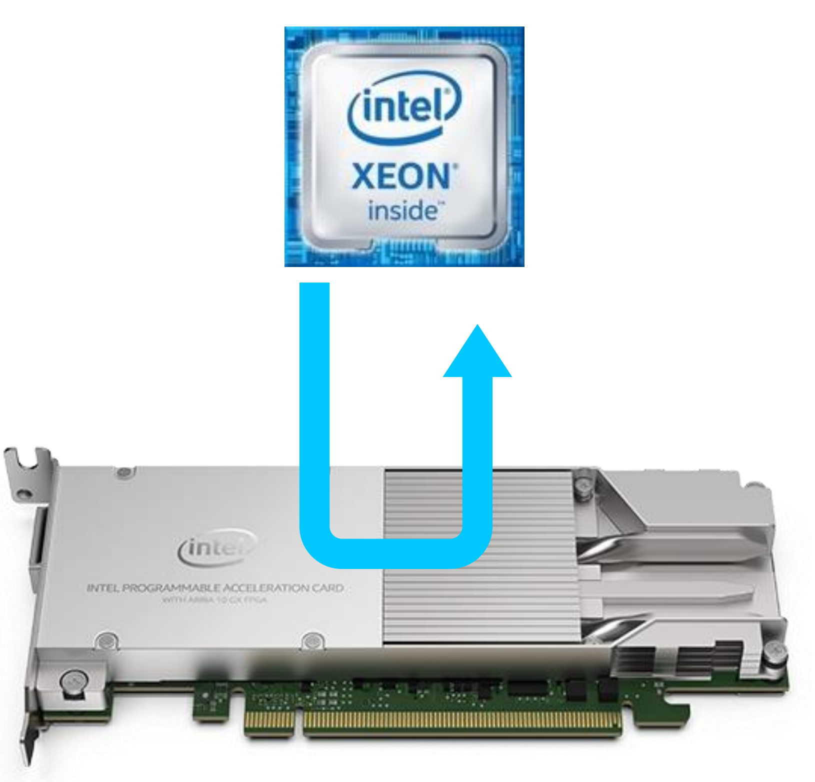
```
dpcpp -fintelfpga main.cpp lib.a
```

Copyright © 2022 Intel Corporation

intel.

FPGA Hardware for oneAPI

FPGA Boards and Board Support Packages



To interface with computers, FPGAs must be mounted onto boards

Boards provide:

- power and thermal management
- memory
- physical interfaces between the FPGA and other devices

Board Support Packages (BSPs) => interface between boards and the Intel oneAPI compiler

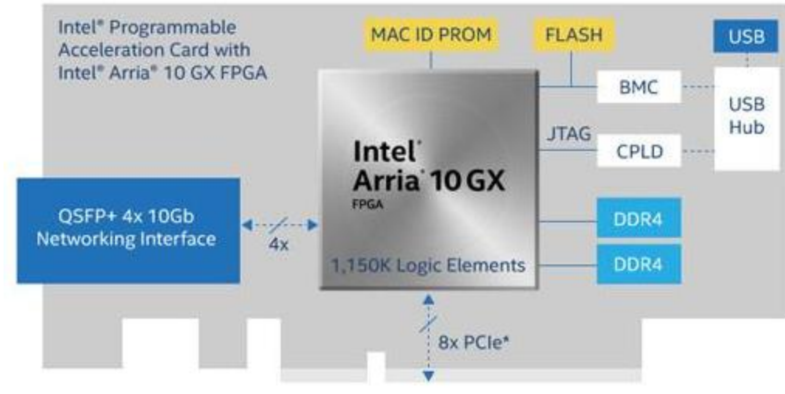
A BSP consists of:

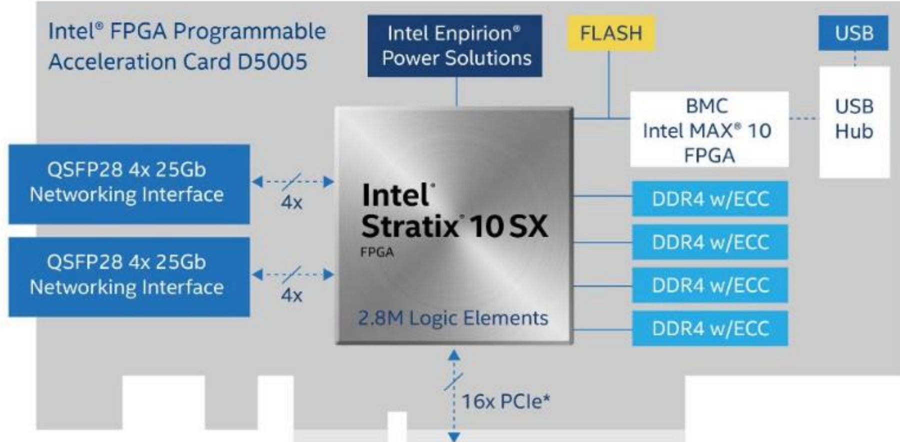
- software layers (card drivers, card management code)
- a precompiled FPGA hardware design implementing memory and physical interfaces and card management logic

The FPGA design implementing the kernel(s) is stitched by the compiler into the framework provided by the BSP

Copyright © 2022 Intel Corporation

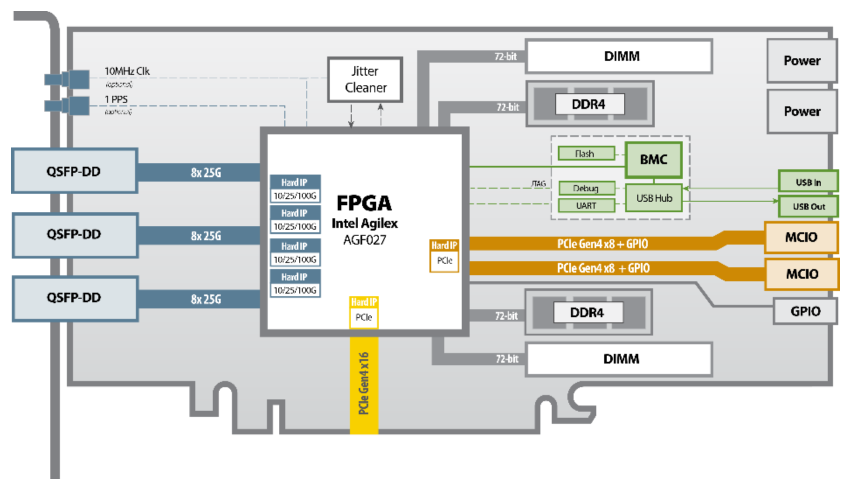
Intel® FPGA Cards Available for use with oneAPI





Copyright © 2022 Intel Corporation

Example of Intel® Agilex® FPGA Card with oneAPI



Bittware®: IA-840F

Copyright © 2022 Intel Corporation

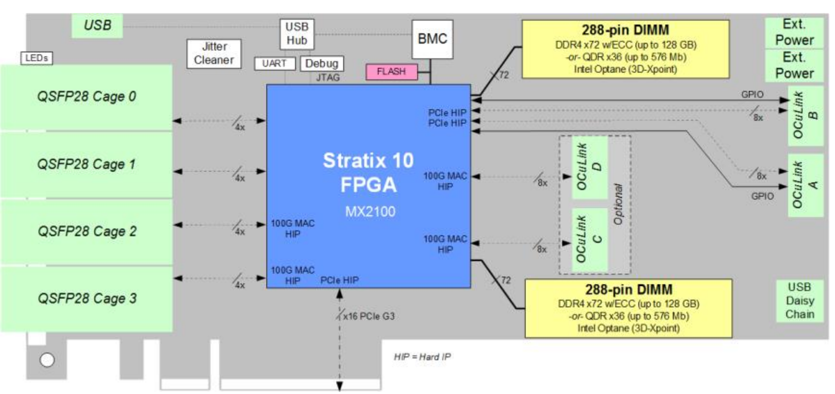
FPGA supported Technologies

FPGA Advantages

- UPI (coming soon: CXL)
  - Cache-coherent, high bandwidth and low latency interface
  - Memory extension with DDR4/DDR-T
- High Bandwidth Memory
  - In package 16GB HBM2, up to 512GB/s
- IKL – FPGA to FPGA connection
- Network interfaces

Intel® S10DX

Bittware®: 520N-MX



Introduction to ntel® DevCloud

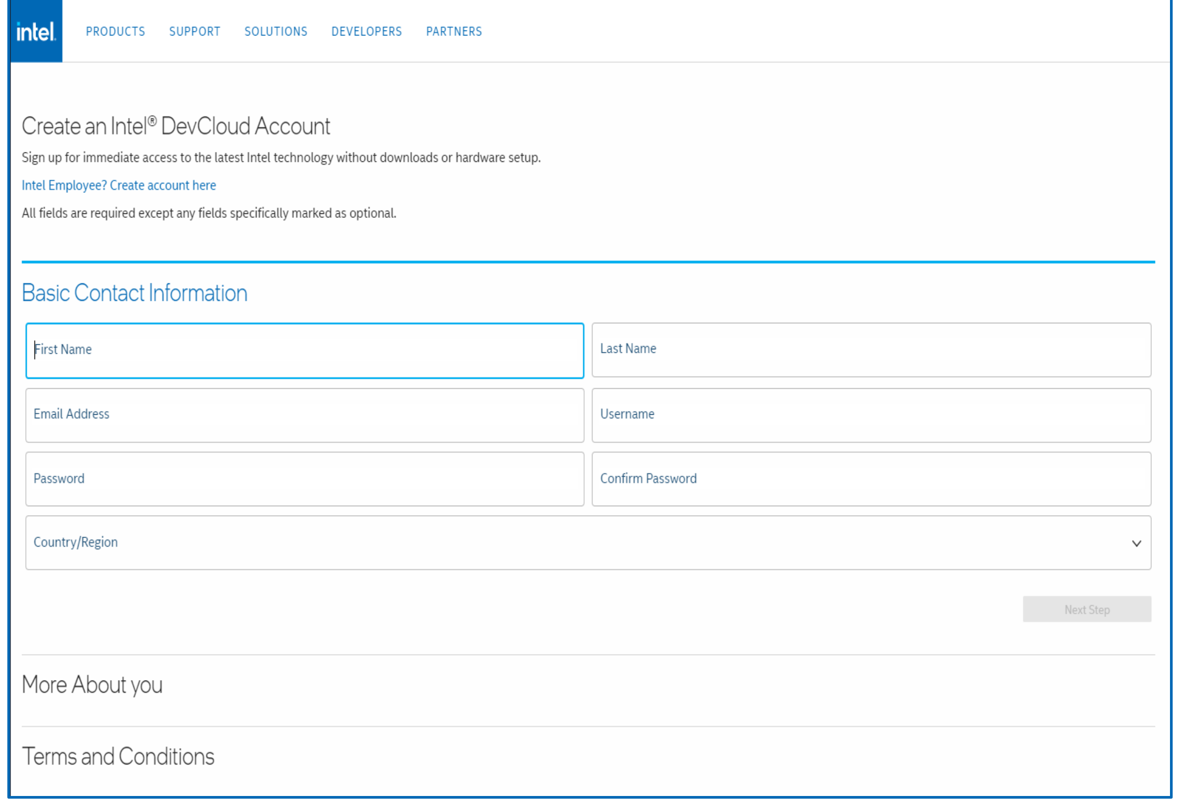
Intel® DevCloud

- Free cloud environment for learning and project prototyping
- Complimentary access to a wide range of Intel® architectures (including FPGAs)
- Pre-installed Intel® optimized frameworks, tools, and libraries

Copyright © 2022 Intel Corporation

Intel® DevCloud

- Sign up here:
  - https://software.intel.com/devcloud
  - Account for 120 days
  - Intel® oneAPI environment pre-installed and ready for use
  - Nodes with cards installed in the group fpga_runtime
  - Nodes with extra memory for full FPGA compiles in the group fpga_compile



Use the Intel® DevCloud

- SSH to gateway
- Or Jupyter Notebooks via browser

- Job queue for compile and run
  - Job output into log file
- Access to single node also possible
- Session time limit:
  default : 6hrs
  max : 24hrs
- Many samples and tutorials in Git repo

https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2BFPGA

Copyright © 2022 Intel Corporation

Intel® DevCloud – Available FPGA Hardware

- What are you trying to use the Devcloud for?

1) Arria 10 PAC - RTL AFU, OpenCL
2) Arria 10 - OneAPI, OpenVINO
3) Stratix 10 - RTL AFU, OpenCL
4) Stratix 10 – OneAPI
5) Emulation
6) Compilation (bitstream creation)

Coming soon: Agilex cards with OneAPI support

Copyright © 2022 Intel Corporation

Summary

FPGA compute acceleration with Intel® oneAPI

- Intel® oneAPI enables software engineers to use easily Intel FPGAs as compute accelerators without deep FPGA knowledge
- Unified programming model for different device types is making port between devices simple
- Modern programming language (DPC++)
- Support of industry standard debug and optimization tools like GDB, Intel® VTune™
- Including optimized RTL blocks as libraries
- Test easily in Intel® DevCloud ☺

- Let your use case also benefit from Intel® oneAPI FPGA compute acceleration

Copyright © 2022 Intel Corporation

intel®